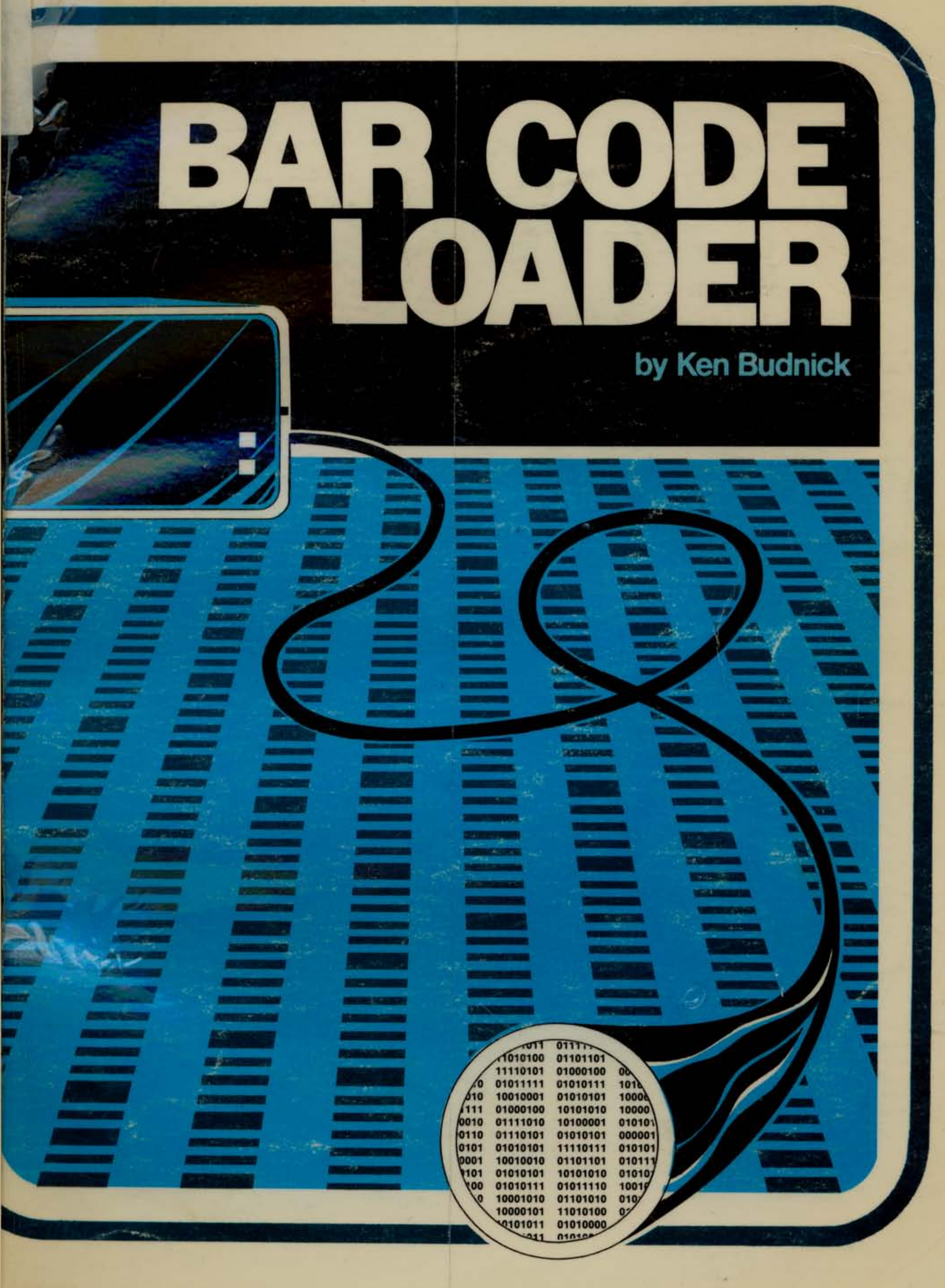
$2.00

A PAPERBYTE™ BOOK

BAR CODE LOADER

by Ken Budnick



PAPERBYTE TM – An Exciting New Way To Distribute Software

One of the most common problems for users and suppliers of personal computer software is the need for product distribution in a form which is helpful to the user, low in cost, tolerant of errors in production use, and free of the need for expensive highly specialized peripherals. One solution, conceived in detail by Walter Banks of the Computer Communications Network Group at the University of Waterloo, Ontario, Canada, is the use of bar code patterns prepared on a computer controlled phototypesetter. A bar code is a linear array of printed bars of varying width which encodes digital data as alternating patterns of black ink and white paper. By using a ruler as a guide, an inexpensive hand held "wand" scanning unit converts the bar patterns into a time varying logic level signal. This time varying binary value can then be interpreted by a program which understands the format of the bars.

The purpose of this pamphlet is to present the decoding algorithm which was designed by Ken Budnick of Micro-Scan Associates at the request of BYTE Publications Inc. The text of this pamphlet was written by Ken, and contains the general algorithm description in flow chart form plus detailed assemblies of program code for 6800, 6502 and 8080 processors. Individuals with computers based on these processors can use the software directly. Individuals with other processors can use the provided functional specifications and detail examples to create equivalent programs.

Paperbyte™ Bar Code Loader

By

Ken Budnick

Credits

Technical Design and Implementation	Ken Budnick
Bar Code Samples	Walter Banks, University of Waterloo
Technical Editor and Producer	Carl Helmers
Production Manager	Christopher Smith
Cover Design	Christopher Smith
Cover Art	Dawson Advertising Concord NH
Production Art	The Hilltop Studio Amherst NH
Printing	Rumford Press Concord NH

Acknowledgements

This book would have been impossible to produce without the technical support of Micro-Scan Associates, P O Box 705, Natick, MA 01760. The Programs contained in this book were designed by Ken Budnick and tested using the input wand and electronics designed by Fred Merkowitz. The labors of these gentlemen have helped advance the state of the art in printed software by several orders of magnitude.

Paperbyte™ Bar Code Loader

Table of Contents

BYTE Publications and Paperbyte ™ Software

notes by Carl Helmers, Editor in Chief

The bar code format presented here was conceived as a result of a telephone conversation between Walter Banks and myself in August of 1976. This conversation led to Walter's presentation on bar code technology at the Personal Computing '76 show in Atlantic City NJ in August 1976. It was Walter who came up with a practical way to implement printed software, a prospect which had been a relatively low priority "wouldn't it be neat if we had a way . . ." kind of idea in our minds before we met.

Our intent is to promote a method for recording machine readable printed software that would be both easy to use and publicly available for software product distribution. We have no intentions of restricting the use of this kind of notation in any way. We believe that its relationship to the personal computer software industry parallels that of written music notation to the music industry: no one company, individual or organization has any specific proprietary claim to the notation itself; rather it is the intellectual property expressed by music notation which is produced and distributed by composers and music publishing companies. (The legal and ethical comparisons between the software publishing and music industries do not stop at this one point.)

As a firm, BYTE Publications Inc does formally claim trademark on our "brand name" of Paperbytes™. I feel that BYTE magazine's articles and software books that use bar code machine readable text have a distinctive quality of style and technical excellence which sets them apart from the ordinary. This pamphlet serves as but one example of our product, the kind of technical documentation and information which is needed by individuals experimenting with the personal use of computers.

Our purpose as a book production company is to make high quality technical documentation of software products available to personal computer experimenters. Mass production allows us to make these products available at relatively low prices when compared with the cost of similar software items in the recent history of the computing industry. Our Paperbytes™ assemblers, compilers, interpreters, operating systems and applications programs come complete with source code listings, relevant object code listings, and machine readable bar code format. Paperbytes™ provides a means by which software artists can earn royalties from their creations by making them available to a larger number of people, thereby benefiting both the author and the computing public. I see this as a technological turning point in the history of computer software.

Carl Helmers

Carl Helmers
BYTE Publications Inc
August 15, 1977

The Bar Code

Bar codes are the newest form of software communication. Combining efficiency of space, low cost, and ease of data entry, bar codes were originally used for product identification in inventory control and supermarket checkout. Because of their direct binary representation of data they are an ideal computer compatible communications media. By using a simple but reliable bar code format and a low cost scanner, the Paperbytes machine readable representation gives the small system user an inexpensive method of input for new software purchased in printed form.

Figure 1 shows how data is coded in bar code format. Binary data is coded in bars of two different widths measured in terms of a unit width. A black bar one unit wide is a zero, while a black bar two units wide is a one. Spaces are also one unit wide.

[In Paperbytes^{TM} books and articles, the physical constraints of the phototypesetting machines currently employed make this unit width 1/72 part of an inch (0.0139 inches, or 0.353 mm). There is nothing sacred about this particular choice of size, since the software used to read the bars is adaptive and only cares about ratios of bar width. . . . CH*]*

The data to be coded is broken into records or frames, where one frame is one line of bars on the printed page. Figure 2 shows the frame format. Each frame can be divided into three parts: header, data, and trailer. The header consists of four bytes and starts with synchronization character (96 hexadecimal) which is used to define the start of the 8 bit byte boundaries within the frame. In addition, this character is used to establish the scanning rate and provide an initial reference in decoding the bars. This is followed by a checksum byte which is the two's complement of the modulo 256 sum of the rest of the header and the data. If the frame is read correctly the sum of the checksum and all following bytes in the frame will be zero. This provides a simple but effective means for the program to determine if any errors have been made in scanning the frame. The next byte is the frame identification. The first frame will have an identification of 0; the second frame's identification will be 1, etc., being incremented by one to the last frame. This identification makes it possible to rescan a line in case of error. As a frame is being scanned, the program can check the identification to see whether this is a rescan of the last frame or a scan of the next frame. The final byte in the header is the frame length, which is a count of the number of data bytes in the data section of the frame. If the length is zero, then the frame is interpreted as an end of file record.

If the file represented in this format requires more than 256 frames, the identification number will wrap around module 256. This number is used solely to establish local order during an input operation, so that the loader can verify an orderly progression of the sequential frames of a long program.

The header section is followed by *n* data bytes, with *n* being the length specified in the header. In present practice the data section has one of two formats depending on the type of data it contains (see figure 3). A text format frame consists of *n* data bytes. This format is used for data which does not have a memory address associated with it. An absolute loader format frame also in current use, has a memory address in the first two bytes of the data section, followed by *n-2* data bytes. This format is used for programs or any other data which must be loaded into specific memory locations.

Finally, the frame ends with a trailer which consists of a single zero bit. This bit is necessary for those decoding schemes which measure the spaces to derive the scanning velocity.

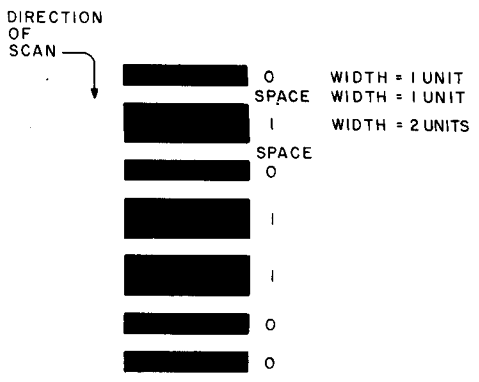


Figure 1: Bar code format. As used in Paperbytes^{TM} products, data is coded using a bar width modulation technique where width is measured in terms of a single unit. In current practice the unit of width is 1/72 part of an inch (0.0139 inches, 0.353 mm). Each bit is represented as a bar followed by a space one unit in width. The zero bar is one unit in width; the one bar is two units in width. Thus the complete pattern of a single bit cell is either two units or three units in width.

Loader Design Considerations

At first glance it would appear that the software to decode bar codes would be quite simple. It would seem that one needs only check the output of the scanner for zeros and ones and then assemble them into 8-bit bytes. Unfortunately, the solution is not quite this simplistic. The software to decode bar codes must be capable of handling many different problems such as speed variation and acceleration, spots and drop-outs, varying print quality, and noise from the scanner. The algorithm design and programs presented here are able to handle all of these problem areas.

One of the more severe problems is speed variation. When using a scanner the average person will vary his scanning rate from about 10 to 40 inches per second (25 to 102 cm per second). Therefore the software must be able to allow for speed variations of several hundred percent. This large speed variation eliminates the possibility of decoding the bars by directly measuring bar widths with respect to a processor clock. Some simple calculations will show that a zero bar at 10 inches per second will be one and one half times as wide as a one bar at 30 inches per second. This is almost a complete reversal of the proper relationship between zeros and ones, where a zero bar should be only half as wide as a one bar.

One possible method for solving this speed variation problem is to compare each bar to the space which follows it. Since all spaces are as wide as a zero bar we now have a reference to use in decoding the bar widths. This method however has several drawbacks. First, since we are timing both bars and spaces there will be no time left over to process data. A 1 MHz processor clock on a typical 8 bit machine is simply too slow to allow long timing loops or the use of interrupts because the counts representing the bar widths would become too small to allow for accuracy. Since data cannot be processed on the fly, it would appear to be necessary to store the raw counts in an intermediate buffer for later processing by another routine in order to arrive at the final data. This not only wastes large amounts of memory but results in a program that is unnecessarily complex.

A different approach to the speed variation problem (and the one used here) is to use "adaptive" software. In this method the program does not know how wide a zero bar (or a one bar) is supposed to be. Instead it knows that the first bar in each frame is a one. One half of the width of this bar is used as a "unit" width (i.e. a zero bar is one unit wide and a one bar is two units wide). The next bar which is scanned is compared to the unit width to determine whether it is a zero or one. Any bar which is less than 1½ times the unit width is considered to be a zero, and any longer bar is a one. In addition, as each bar is read, its width (in the case of a one bar, half its width) is averaged with the unit width to arrive at a new unit width to use in decoding the next bar. This method assumes that the speed will not change drastically in two bar widths, which is a valid assumption under normal scanning conditions. If the scanner is used with a light touch so that it does not stick and jump as it moves across the page the software will be able to handle most of the speed variations that are likely to occur.

Since this method does not measure the spaces it is possible to do the processing for each bit during the space that follows it. This allows the data to be decoded immediately and stored in its final location in memory without the use of intermediate buffers or post-processing. This results in a shorter and simpler program, a program which does not require a large memory buffer for input processing.

A second problem, closely related to speed variation, is acceleration. This problem occurs in two different forms. First is the acceleration as the operator begins moving the scanner at the beginning of the frame. If the operator normally scans at around 30 inches per second, it would be necessary to accelerate from 0 to 30 inches per second in a fairly short distance. This requirement is not too severe, so the problem can be largely eliminated with a "running start". When used properly, the scanner should be placed at least one inch away from the first bar in the frame, then most of the acceleration will occur before the first bar is detected. When reading Paperbytes^{TM} bar codes with the programs presented here, it is possible to read right over the humanly readable print of the frame number and relative data address. This "invalid data" appearing at the beginning of each frame is ignored, because the program is seeking a synchronization character pattern. This should give a more than adequate margin for acceleration. Similarly, deceleration (and thereby slow speed) at the end of the line is a potential problem. The solution here is to follow through. Scan right off the end of the frame. This will insure that the large decelerations occur after reading the last bar in the frame. In the printed form, Paperbytes^{TM} bar codes are positioned with ample acceleration and deceleration zones at the top and bottom of the page.

The second area where the problem of acceleration (and deceleration) occurs is when the scanner sticks and jumps as it moves across the page. This problem is so severe that no scanner or software in the world could take care of it. Luckily, the solution here is also quite simple. In our experience, this problem is caused by using excessive pressure when scanning the page. All that is required is enough pressure to insure that the scanner does not lift away from the page in the middle of a frame.

Another common mistake is to grip the scanner too tightly. This makes it difficult to maintain a light pressure against the page. The correct procedure is to grasp the scanner lightly with the finger tips, keeping everything from the fingers to the shoulder loose and flexible. When the scanner is used in this manner it will seem to "float" across the page, with a nice *even* pressure and speed.

Another problem which must be handled by the scanning program is the presence of spots during the white spaces and dropouts during the bars. The spot problem is relatively minor because during much of the space the software is not looking at the scanner output because it is busy processing the last bar. Therefore it never sees any spots which occur in the first part of the space. Later spots are handled in the same manner as dropouts. The dropout problem is more severe because the program will see all the dropouts which occur. To help eliminate this problem software filtering has been included. Since a spot will appear to be a very short bar, each bar is required to be at least one fourth of the unit width. Similarly, a dropout will appear as a short space. Therefore, when a space is detected, a short loop is entered to assure that the space has a certain minimum width. Otherwise it is considered to be a dropout. Bar widths are accumulated until the total width is greater than one fourth of a unit width and a minimum width space is detected. At this point the program has read a valid bar and begins processing it.

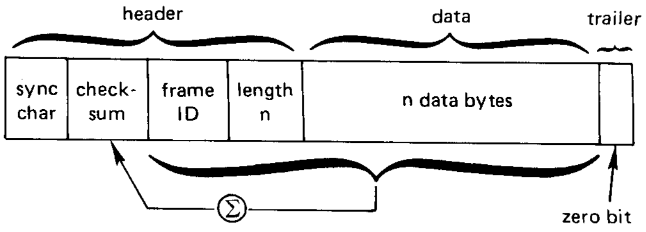


Figure 2: Frame Format. (a) The frame is divided into three major sections. The header section contains four bytes (8 bit) of overhead information. It begins with a synchronization character (hexadecimal 96). This is followed by a checksum of the remaining bytes in the frame. The frame Identification byte is a sequential 8 bit integer used to keep track of the order of frames. The length byte specifies how many data bytes are contained in the balance of the frame. The data section contains "n" 8 bit data bytes where n is the value of the length byte in the header. The trailer consists of a single zero bit used to define the space following the last bit cell in the frame.

A) Synchronization pattern hexadecimal 96

B) Check sum hexadecimal EC

C) Line identification, hexadecimal 2D, decimal 45

D) Length, hexadecimal 1C, decimal 28

E) Data field, 28 bytes with the following values:

```
05  B5  BF  70  15  04  CC  70
BC  04  D1  70  BE  04  D4  FF
74  04  D7  FE  4B  04  DB  70
BC  04  E0  70
```

F) Single zero width bar as trailer.

(b) A single bar code frame taken from a typical Paperbytes^{TM} product illustrates this format. The bytes of this frame are listed to illustrate a specific example. This frame was created by Walter Banks at the University of Waterloo, and is taken from the object text of a 6800 processor program called MONDEB written by Don Peters of Nashua, NH.

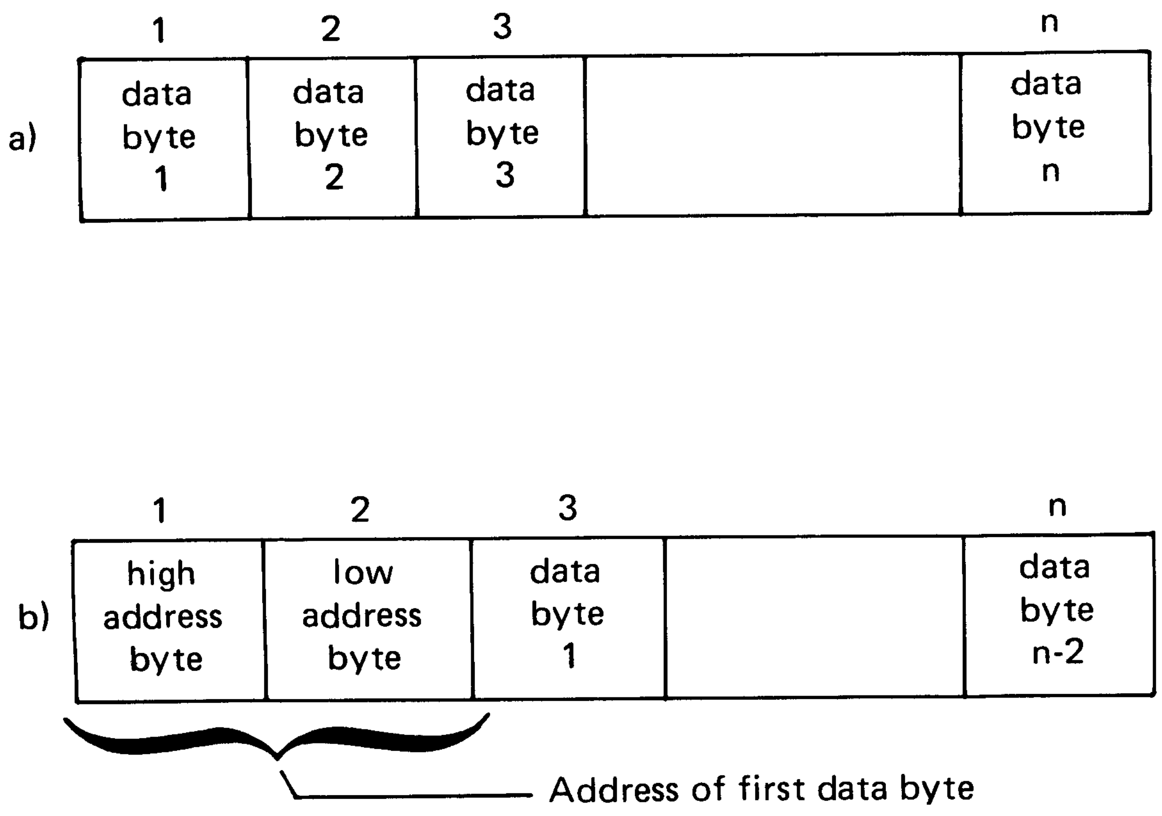


Figure 3: In current Paperbytes^{TM} software products, two formats for the data field of a frame of bar codes have been used. The most common practice is to use a text format data field as shown in (a). Here the optical bar code medium is being used to transfer an address independent block of text into the user's computer for later processing according to the specific needs of the software involved. This form is intended for character texts as well as object code data input to relocation schemes. A second data field format currently in use is shown in (b). This absolute loader format is used for data which will be loaded in a known segment of address space at addresses contained in the first two bytes of each frame.

A General Bar Code Loader Algorithm

In this publication I've provided a set of three bar code loader programs appropriate for use with Paperbytes^{TM} software products and articles appearing in BYTE magazine. The detailed programs are written and assembled for the 6800, 6502 and 8080 microprocessor designs.

All three programs presented here use the same general algorithm for reading the bar codes. Figure 4 shows a high level flow chart which applies to all programs. The algorithm has been divided into four subroutine to make it easier to understand and modify. The first is the main or control subroutine. This calls the other three to decode the bytes, separates the header bytes, and then stores the data bytes into memory. The second subroutine reads one byte from the bar codes and adds it to the checksum. The third subroutine reads a single bit of data. And the fourth subroutine reads the length of a bar. The operation of these subroutines will be more easily understood if they are studied in reverse order.

LDA, LDR Subroutine

The last subroutine is the control loop. It contains two entry points: LDA, which loads absolute data, and LDR, which loads relocatable data. The only difference between the two entry points is the setting of the text or absolute format indicator flag. The LDA entry sets the flag to a "1" and the LDR entry sets it to a "0". Next, ID (the frame number of the frame being scanned) is initialized to 0. At LD4 the timing bit is read by calling RBAR. Since the timing bit is a one, its length must be divided in half to arrive at the UNIT width (this timing bit is actually the first bit of the synchronization character). The header is now read and values are saved for later use. At LD6 a loop is entered to search for the rest of the synchronization byte (hexadecimal 16). This is done by calling RBIT to read bits until the assembled BYTE equals 16 hex. Next, at LD8, the checksum (CKSM) is read and saved. At LD10 the frame number is read and compared to ID (the identification number of the last frame scanned). If the frame number equals the identification number a rescan of the last frame is implied. It is therefore necessary to reset the buffer address pointer to the value it had at the beginning of the frame the last time. This value was saved in ABUF. If the frame number equals ID plus one, then the next frame is being scanned. The new frame number is saved in ID and ABUF is set to the present value of the buffer address pointer (in case this frame is rescanned). If the frame number has any other value then an error has occurred and control is transferred to LD4 to prepare to read another frame. Next, at LD14 the frame length (LEN) is read and saved. If LEN = 0 then this is an end-of-file frame and if the CKSM is zero then control is returned to the user. If LEN is not zero then there is data to be read. If flag is zero, then this is text data and the program skips to LD18 to read the data. However if flag = 1, then it is absolute data, and the address of where to store the data is contained in the first two bytes of the data section. This address is read by two calls to RBYT and saved in the buffer address pointer. (Note that the previous process of saving and/or retrieving a buffer address from ABUF has meaning only for a text format frame. However, the process is carried out for both text and absolute types in order to simplify the program.) Finally at LD18 a loop is entered to read and store the data bytes. When all data bytes have been read, the CKSM is checked. If it equals zero then the frame has been read correctly and the bell on the terminal is rung as an indicator (ASCII hexadecimal value 07). Control is then transferred to LD4 to prepare for reading the next frame.

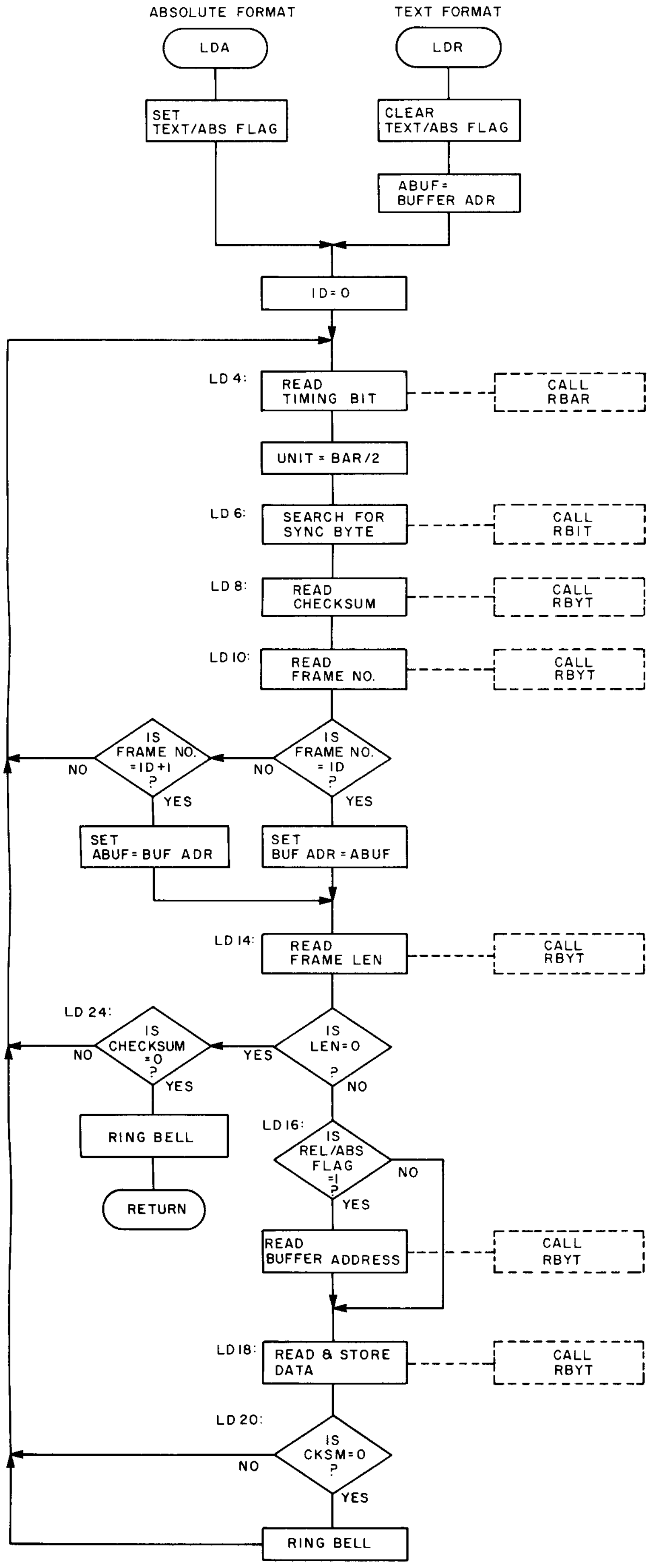
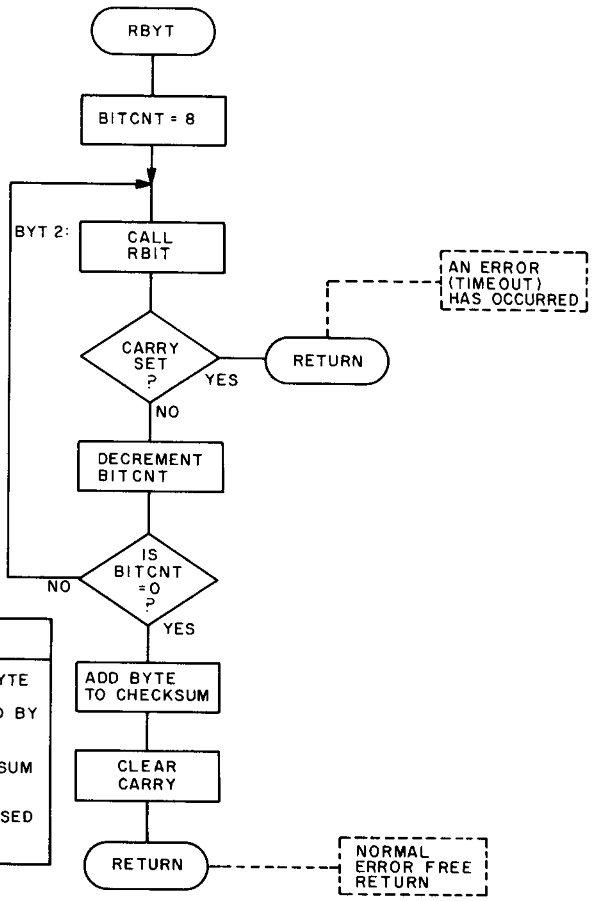


Figure 4a: The main program of the bar code loader software. Two entry points are defined. LDA sets FLAG=1 to indicate use of the absolute loader format defined in figure 3b. LDR clears FLAG to indicate loading of a block of text starting at the initialized value of ABUF. The lower level subroutines RBAR, RBIT and RBYT are called by this routine from the points noted. Labels of the form LDN show corresponding points in the detail assemblies of listings 1, 2, and 3.

RBYT Subroutine

The RBYT (Read Byte) subroutine reads an 8 bit byte. This is accomplished by calling RBIT eight times. If RBIT returns an end of frame timeout indication (carry flag set), RBYT immediately returns to the calling routine with the carry flag still set. When the entire byte has been read it is added to the checksum. The checksum was of course initialized to zero for the line identification prior to the beginning of the RBYTE call.) Finally the carry flag is cleared to indicate that a byte has been read and RBYT returns to the calling routine.

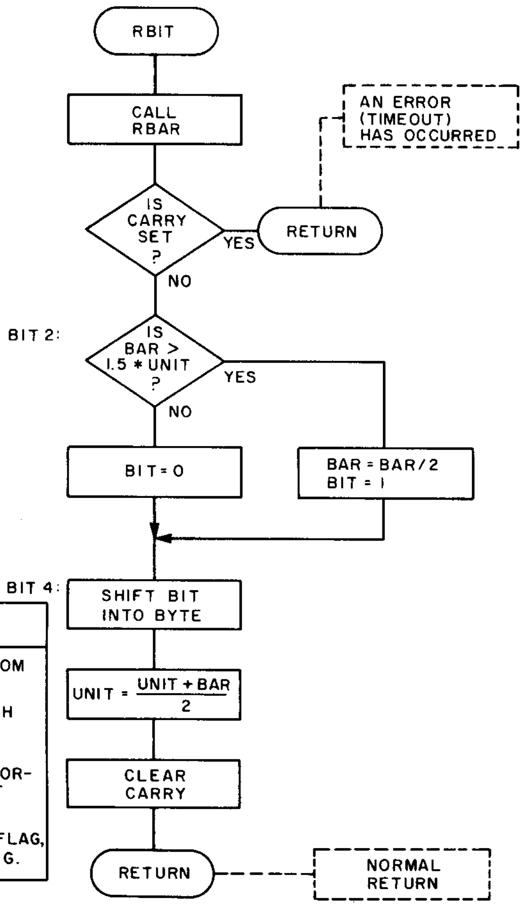


VARIABLES	
BITCNT	= COUNTER FOR BITS PER BYTE
BYTE	= 8 BIT INPUT FIELD (LOADED BY RBAR WITH SHIFT.
CHECKSUM	= CURRENT RUNNING CHECK SUM TOTAL
CARRY	= PROCESSOR CARRY FLAG USED AS ERROR FLAG

Figure 4b: The byte read subroutine, RBYT. This subroutine assembles one 8 bit byte of data and adds it to the checksum. Each bit of the byte is read with a call to the subroutine RBIT.

RBIT Subroutine

The RBIT (Read Bit) subroutine reads a single data bit. It starts by calling RBAR to get the width of the bar. If the carry flag is set on the return from RBAR, an end of frame timeout has occurred and RBIT returns to the calling routine with the carry flag still set. If a bar was read, it is compared to the current unit width to determine whether it represents a 0 or 1 bit. Any bar which is less than one and one half unit widths is called a 0 bit and all others are called 1 bits. This bit is then shifted into the low order bit position of the BYTE that is being read. The bar width is then used to compute a new unit width by dividing the bar width in half if it was determined to be a one bit. The bar width is then averaged with the old unit width to arrive at the new unit width and finally, the carry flag is cleared to indicate that a bit was read and RBIT returns to the calling routine. Note that when implementing the algorithm, dividing by one half is done using a right shift operation; calculating 1.5 times a small integer is similarly done with a single bit shift followed by an addition.



VARIABLES:	
BAR	= BAR WIDTH COUNT FROM RBAR
UNIT	= CURRENT UNIT WIDTH
BIT	= OUTPUT BIT, 0 OR 1
BYTE	= OUTPUT BYTE (LOW ORDER HAS LATEST BIT SHIFTED INTO IT)
CARRY	= PROCESSOR CARRY FLAG, USED AS ERROR FLAG.

Figure 4c: The bit read subroutine, RBIT. This subroutine decodes a single bit of data and shifts it into the BYTE which is being assembled. This subroutine contains the adaptive portion of the program which eliminates dependence upon speed and acceleration by averaging each new BAR width with the previous UNIT width. Each bar width is measured using the subroutine RBAR.

RBAR Subroutine

The RBAR (Read Bar) subroutine returns the width of a single bar. It includes filtering to eliminate spots and dropouts and, if there is no change in the scanner output for a long period of time relative to a typical bandwidth, returning an end of frame timeout indication. The subroutine measures the bar width by incrementing a counter in a timing loop. Thus the bar width is a count in the range of 0 to 255.

The program actually keeps two counters, one for spaces and another for bars. The only use of the space counter is in detecting the end of a frame. If either counter overflows, the program assumes that the end of the frame has been reached and returns an end of frame timeout indication to the calling routine.

The RBAR subroutine consists of three timing loops starting at BAR2, BAR4, and BAR6. The first loop (at BAR2) cycles until a bar is detected, at which time the space counter is incremented. When a bar is detected, the second timing loop (at BAR4) is entered. This loop increments the bar counter until a space is detected. The bar width is now checked to see if it is greater than one fourth of the current unit width. If it is not, this bar is assumed to be a partial bar (caused by a dropout) and the first timing loop (BAR2) is reentered to wait for the rest of the bar to be detected. If the bar width is greater than one fourth of the unit width, the third loop (at BAR6) is entered to make sure that the space has a certain minimum width. If the space is too short, it is assumed to be a dropout in the bar and the second timing loop (BAR4) is reentered to continue reading the bar. Finally, when this trailing space is found to be wider than the minimum width, the subroutine clears the processor's carry flag to indicate that a bar has been read and returns to the calling routine. If a counter overflows in any timing loop, the subroutine sets the carry flag to indicate an end-of-frame timeout before returning. (The carry flag is thus used as an error indicator.)

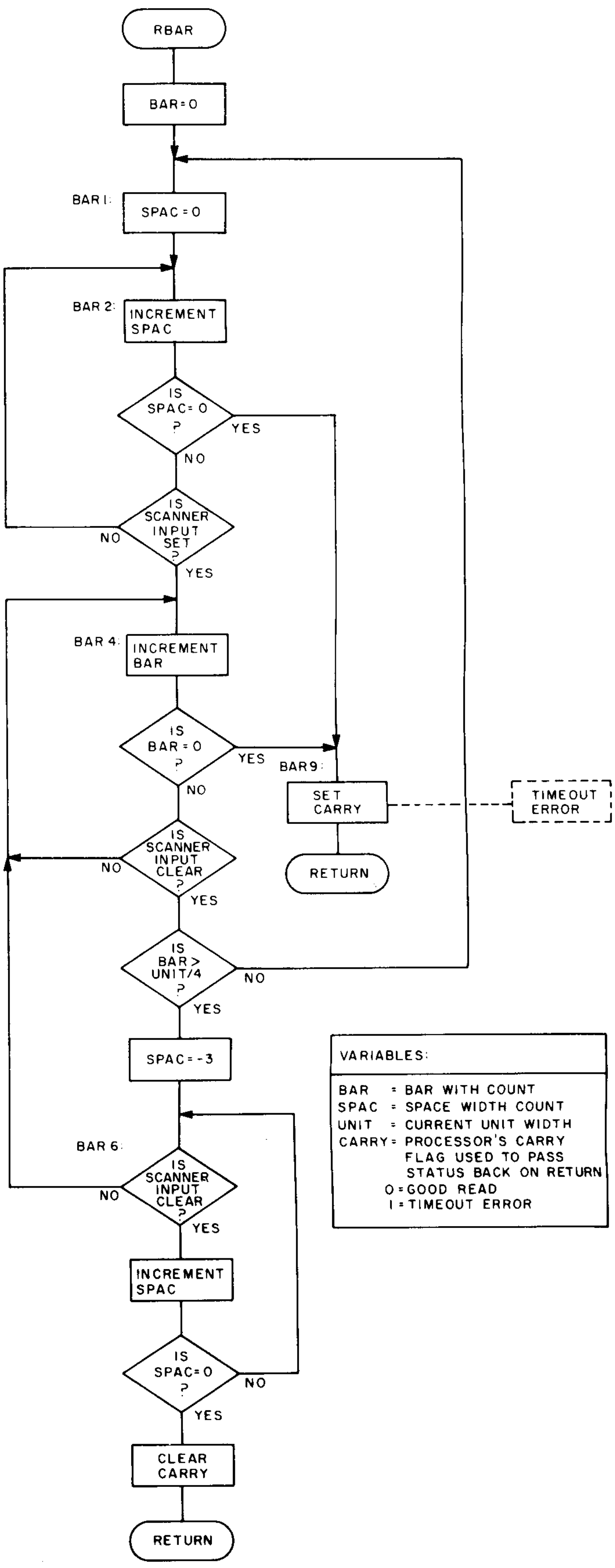


Figure 4d: The bar width measurement subroutine, RBAR. This subroutine times the width of a single bar of data input from the scanner. A bar starts when the scanner input becomes logical 1, and it ends when the scanner input again becomes logical 0. Filtering for dropouts and ink blotches is provided by testing to make sure that the measurement is greater than the current UNIT width divided by 4.

Adjusting Program Timing Loops

While the program of listing 1 is address independent due to the use of relative addressing on all branches, several assumptions have been made about the hardware address commitments of the system which uses the program. All the hardware address space commitments are essentially arbitrary, and should be changed to reflect the characteristics of the 6800 system in which this code is actually used.

The origin of hexadecimal 1000 for the program itself was arbitrarily chosen as a "nice" round number that is far away from page 0. In order to take advantage of direct addressing, all scratch data areas of the program have been assembled at locations hexadecimal 30 to 36 in page 0. These locations can be changed by hand to any location within page zero by modifying each use within the listing, or with re-assembly using the source code of listing 1. The data areas can be reassembled anywhere in memory if desired, using extended addressing instead of direct addressing of page 0, but some thought should be given to the effect this will have on the execution time characteristics of the program.

The program also assumes that the user has a simple 8 bit input port wired to hexadecimal address 8000 such that the high order bit of the port reads the value of the scanner's output: logical level 1 for input of a bar opposite the scanner's aperture, and logical level 0 for input of a space under the aperture. This port must be initialized prior to entry into the scanning routine, so users of PIA ports should do this either by hand or using a program set up the proper PIA configuration for input.

An ASCII "bell" character output is used as operator feedback to indicate end of frame without error. This program assumes a Motorola MIKBUG monitor program with a character output routine located at hexadecimal address E1D1.

Unlike the 6800 program of listing 1, the 6502 program is not address independent. An origin of hexadecimal 300 was chosen for the program based on the original system's characteristics. The 6502 system used for this version's testing is reflected in the choice of the location for a routine to type out a single ASCII character at location 02D9, and the input port which is assumed to be located at hexadecimal address FC12.

The program timing loops in RBAR must be set up so that the resulting counts do not get too small on zero bars when scanning fast, or too large on one bars when scanning slow. If the computer is slow (or the timing loop too long) then accuracy will decrease resulting in more errors. This will force the user to scan at a slower rate. If the computer is fast (or the timing loop too short) then the counts will overflow at slower scanning speeds causing end of frame timeouts to occur. This will force the user to scan at a higher speed, which significantly increases the wear on the page of bar codes. Table 1 shows the time required to scan zero and one bars at various scanning rates. The table also gives the counts that would result from a 16 μs timing loop. (This count is found by dividing the given times by the length of the timing loop in microseconds.) For good accuracy, a zero bar scanned at the highest speed should give a count greater than 20 and a one bar scanned at the slowest speed should give a count less than 200. If the loader program does not seem to work reliably on your system, calculate these counts for the timing loop at BAR4. If the counts are too high, then insert some NOPs or other "do nothing" instructions into each of the timing loops to slow them down. If the counts are too low, then either the computer or the timing loops will have to be speeded up, or you should scan the bars more slowly.

		Scanning Rate		
		10 ips	20 ips	30 ips
Data Bit Value	zero bar (.014 in)	1400 μs/87	700 μs/43	466 μs/29
	one bar (.028 in)	2800 μs/175	1400 μs/87	932 μs/59

Table 1: Time and counts required to scan a bar at various rates of speed. In each position of the matrix, the number to the left of the slash is the number of microseconds that a bar will take in crossing the scanner head at a given rate of scan. The number to the right of the slash gives the integer width count for the bar, assuming a (typical) 16 μS timing loop performs the measurement.

The 6800 Bar Code Loader Program

The 6800 program of listing 1 uses the A, B, and X registers to hold the checksum, decoded byte, and storage address. Locations 0030 through 0036 hold the other program variables to allow direct addressing. The program uses relative addressing only for branches. This means that it can be loaded anywhere in memory without modification and will still operate correctly provided that the destination storage address does not overlap the program's location.

This program was developed on a SWTP 6800 which runs at a processor clock rate which is a little less than 1 MHz. The efficiency of the 6800 resulted in timing loops which were much too fast, therefore they had to be almost doubled in length. This was accomplished simply by repeating the TST instructions a number of times. The redundant TST instructions have the comment "KILL TIME" to indicate their use. A total of 12 processor states per loop are wasted with two TST instructions in listing 1. By removing these redundant instructions the program will operate reliably even on 500 kHz systems. If you are fortunate enough to have one of the newer 6800 chips running at 1.5 MHz or 2 MHz then additional time wasting instructions will be necessary to slow the timing loops down even more.

LISTING NO. 1

```
*    ---------
*    LDA , LDR
*    ---------
*
*
*     SUBROUTINES TO LOAD DATA FROM BAR CODE SCANNER
*     INTO MEMORY.
*
*     LDA - LOADS ABSOLUTE BINARY DATA INTO MEMORY.
*           MEMORY ADDRESS IS CONTAINED IN DATA FRAME.
*
*     LDR - LOADS RELOCATABLE (E.G. ASCII) DATA NOT
*           ASSOCIATED WITH A MEMORY ADDRESS.
*           ENTER WITH X REGISTER CONTAINING ADDRESS
*           OF WHERE TO STORE DATA.
*
*     REGISTER USAGE:
*
*          A - CHECKSUM
*          B - DECODED BYTE
*          X - STORAGE ADDRESS
*
*          A AND B REGISTERS ARE SAVED ON ENTRY AND
*          RESTORED ON EXIT.  X WILL CONTAIN ADDRESS
*          OF LOCATION AFTER LAST DATA BYTE LOADED
*          INTO MEMORY.



      OPT  O
      OPT  S
      ORG  $1000

SCNR EQU  $8000     SCANNER ADDRESS
TYPE EQU  $F1D1     ADDR OF ROUTINE TO TYPE A CHAR

ABUF EQU  $30       BUFFER ADDR AT BEGINNING OF FRAME
ID   EQU  $32       FRAME ID
LEN  EQU  $33       FRAME LENGTH
UNIT EQU  $34       LENGTH OF A ZERO BAR
BAR  EQU  $35       LENGTH OF BAR BEING SCANNED
FLAG EQU  $36       ABS/REL FLAG
```

```
1000  36         LDA  PSHA          ABSOLUTE LOADER ENTRY POINT
1001  86 01           LDAA =1
1003  20 02           BRA  LD2

1005  36         LDR  PSHA          RELOCATABLE LOADER ENTRY POINT
1006  4F              CLRA

1007  97 36      LD2  STAA FLAG
1009  37              PSHB
100A  7F 0032         CLR  ID       INIT FRAME ID
100D  DF 30           STX  ABUF     SAVE BUF ADR

100F  86 40      LD4  LDAA =$40     READ TIMING BIT
1011  97 34           STAA UNIT
1013  8D 6E           BSR  LD26
1015  25 F8           BCS  LD4
1017  96 35           LDAA BAR           UNIT = 1/2 TIMING BIT
1019  44              LSRA
101A  97 34           STAA UNIT

101C  5F              CLRB          SEARCH FOR SYNC BYTE
101D  8D 74      LD6  BSR  RBIT
101F  25 EE           BCS  LD4
1021  C1 16           CMPB =$16
1023  26 F8           BNE  LD6

1025  8D 5E      LD8  BSR  RBYT     READ CHKSUM
1027  25 E6           BCS  LD4
1029  17              TBA

102A  8D 59      LD10 BSR  RBYT     READ ID
102C  25 E1           BCS  LD4
102E  D1 32           CMPB ID
1030  27 0A           BEQ  LD12
1032  5A              DECB
1033  D1 32           CMPB ID
1035  26 D8           BNE  LD4
1037  7C 0032         INC  ID       .    NEXT FRAME
103A  DF 30           STX  ABUF
103C  DE 30      LD12 LDX  ABUF     .    RESCAN

103E  8D 45      LD14 BSR  RBYT     READ FRAME LENGTH
1040  25 CD           BCS  LD4
1042  D7 33           STAB LEN
1044  27 31           BEQ  LD24

1046  D6 36      LD16 LDAB FLAG     SEE IF ABS OR REL
1048  C1 00           CMPB =0
104A  27 14           BEQ  LD18     .    IF REL
104C  8D 37           BSR  RBYT     .    IF ABS - READ ADDRESS
104E  25 BF           BCS  LD4
1050  D7 30           STAB ABUF
1052  8D 31           BSR  RBYT
1054  25 B9           BCS  LD4
1056  D7 31           STAB ABUF+1
1058  DE 30           LDX  ABUF
105A  7A 0033         DEC  LEN
105D  7A 0033         DEC  LEN

1060  8D 23      LD18 BSR  RBYT     READ DATA
1062  25 AB           BCS  LD4
1064  E7 00           STAB 0,X
```

```
1066  08                 INX
1067  7A 0033            DEC   LEN
106A  26 F4              BNE   LD18

106C  81 00        LD20  CMPA  =0       CHECK CHECKSUM
106E  26 9F              BNE   LD4      .     IF ERROR
1070  86 07              LDAA  =7       .     OUTPUT 'CORRECT' SIGNAL
1072  BD E1D1            JSR   TYPE
1075  20 98              BRA   LD4

1077  81 00        LD24  CMPA  =0       EOF READ
1079  26 94              BNE   LD4      .     IF CHECKSUM ERROR
107B  86 07              LDAA  =7       .     OUTPUT 'CORRECT' SIGNAL
107D  BD E1D1            JSR   TYPE
1080  33                 PULB           .     RETURN
1081  32                 PULA
1082  39                 RTS

1083  20 2E        LD26  BRA   RBAR

                   *   ----
                   *   RBYT
                   *   ----
                   *
                   *
                   *     READ ONE BYTE FROM SCANNER
                   *     ADD BYTE TO CHECKSUM
                   *
                   *     EXIT: C(B) = BYTE
                   *           C(A) = CHECKSUM
                   *           CARRY = CLR IF BYTE READ
                   *                 = SET IF END-OF-FRAME TIMEOUT


1085  36           RBYT  PSHA           SAVE A
1086  86 08              LDAA  =8       SET BIT COUNT

1088  8D 09        BYT2  BSR   RBIT     READ BYTE
108A  25 25              BCS   BIT9
108C  4A                 DECA
108D  26 F9              BNE   BYT2

108F  32                 PULA           ADD BYTE TO CHECKSUM
1090  1B                 ABA
1091  0C                 CLC
1092  39                 RTS
                   *   ----
                   *   RBIT
                   *   ----
                   *
                   *
                   *     READ ONE BIT FROM SCANNER
                   *
                   *     EXIT: C(B) = BYTE WITH BIT SHIFTED IN
                   *           CARRY = CLR IF BIT READ
                   *                 = SET IF END-OF-FRAME TIMEOUT


1093  36           RBIT  PSHA           SAVE A
1094  8D 1D              BSR   RBAR     READ BAR
1096  25 19              BCS   BIT9

1098  96 34        BIT2  LDAA  UNIT     SEE IF BAR > 1.5*UNIT  (A ONE BIT)
109A  44                 LSRA
109B  9B 34              ADDA  UNIT
109D  90 35              SUBA  BAR
109F  2A 03              BPL   BIT4
10A1  74 0035            LSR   BAR      ONE BIT - DIVIDE BAR LENGTH IN HALF

10A4  48           BIT4  ASLA           SHIFT BIT INTO BYTE
10A5  59                 ROLB

10A6  74 0034            LSR   UNIT     COMPUTE NEW UNIT
10A9  96 35              LDAA  BAR
10AB  44                 LSRA
10AC  9B 34              ADDA  UNIT
10AE  97 34              STAA  UNIT

10B0  0C                 CLC            RETURN
10B1  32           BIT9  PULA
10B2  39                 RTS
                   *   ----
                   *   RBAR
                   *   ----
                   *     READ BAR LENGTH
                   *
                   *     EXIT: C(BAR) = BAR COUNT
                   *           CARRY = CLR IF BAR READ
                   *                 = SET IF END-OF-FRAME TIMEOUT
10B3  36           RBAR  PSHA           SAVE A
10B4  7F 0035            CLR   BAR      CLEAR BAR COUNT

10B7  4F           BAR1  CLRA           CLEAR SPACE COUNT

10B8  4C           BAR2  INCA           WAIT FOR SCANNER INPUT SET
10B9  27 30              BEQ   BAR9
10BB  7D 8000            TST   SCNR     .     KILL TIME
10BE  7D 8000            TST   SCNR     .     KILL TIME
10C1  7D 8000            TST   SCNR
10C4  2A F2              BPL   BAR2

10C6  7C 0035      BAR4  INC   BAR      WAIT FOR SCANNER INPUT CLEAR
10C9  27 20              BEQ   BAR9
10CB  7D 8000            TST   SCNR     .     KILL TIME
10CE  7D 8000            TST   SCNR     .     KILL TIME
10D1  7D 8000            TST   SCNR
10D4  2B F0              BMI   BAR4

10D6  96 34              LDAA  UNIT     SEE IF BAR > UNIT/4  (VALID DATA)
10D8  44                 LSRA
10D9  44                 LSRA
10DA  90 35              SUBA  BAR
10DC  2A D9              BPL   BAR1

10DE  86 FD              LDAA  =$FD     CHECK FOR SPACE STILL PRESENT
10E0  7D 8000      BAR6  TST   SCNR
10E3  2B E1              BMI   BAR4
10E5  4C                 INCA
10E6  26 F8              BNE   BAR6

10E8  0C                 CLC            NORMAL RETURN
10E9  32                 PULA
10EA  39                 RTS

10EB  0D           BAR9  SEC            END-OF-FRAME TIMEOUT RETURN
10EC  32                 PULA
10ED  39                 RTS

                         END
```

Notes

The 6502 Bar Code Loader Program

The 6502, because it lacks enough registers in the processor itself, must save virtually all program variables in memory. The only exception is the Y index register which is used to hold the decoded byte. All other variables are stored in page zero locations 0030 through 003A. This program was developed for a home brew 6502 system running at 1 MHz. Because of the speed of the 6502 it was necessary to almost double the length of the program timing loops. This was done by repeating the BIT instructions several times (not necessarily the best method). If the redundant instructions are removed the program will run reliably on a 500 kHz system. This program was hand assembled, with listing 2 created using a text editor running on the 6800 system. The hand prepared assembly format of listing 2 uses conventions of a typical 6502 assembler, but has never been actually assembled and could conceivably contain one or more syntax errors of a relatively trivial nature. The object code shown in listing 2 has been successfully executed as it appears here.

LISTING NO. 2

```
*    ---------
*    LDA , LDR
*    ---------
*
*
*
*     SUBROUTINES TO LOAD DATA FROM BAR CODE SCANNER
*     INTO MEMORY.
*
*     LDA - LOADS ABSOLUTE BINARY DATA INTO MEMORY.
*           MEMORY ADDRESS IS CONTAINED IN DATA FRAME.
*
*     LDR - LOADS RELOCATABLE (E.G. ASCII) DATA NOT
*           ASSOCIATED WITH A MEMORY ADDRESS.
*           ENTER WITH ADR,ADR+1 CONTAINING ADDRESS
*           OF WHERE TO STORE DATA.
*
*     REGISTER USAGE:
*
*        A -
*        X -
*        Y - DECODED BYTE
*
*        A, X, Y REGISTERS ARE SAVED ON ENTRY AND
*        RESTORED ON EXIT.  ADR,ADR+1 WILL CONTAIN
*        ADDRESS OF LOCATION FOLLOWING LAST BYTE
*        LOADED INTO MEMORY
SCNR  EQU  $FC12    SCANNER ADDRESS
TYPE  EQU  $02D9    ADDR OF ROUTINE TO TYPE A CHAR

ADR   EQU  $30      BUFFER ADDRESS
ID    EQU  $32      FRAME ID
LEN   EQU  $33      FRAME LENGTH
UNIT  EQU  $34      LENGTH OF A ZERO BAR
BAR   EQU  $35      LENGTH OF BAR BEING SCANNED
SPAC  EQU  $36      LENGTH OF SPACE BEING SCANNED
ABUF  EQU  $37      VALUE OF ADR AT BEG OF FRAME
CKSM  EQU  $39      CHECKSUM
FLAG  EQU  $3A      ABS/REL FLAG

0300  48          LDA   PHA             ABSOLUTE LOADER ENTRY POINT
0301  A9 01             LDA  #1
0303  D0 03             BNE  LD2

0305  48          LDR   PHA             RELOCATABLE LOADER ENTRY POINT
0306  A9 00             LDA  #0

0308  85 3A       LD2   STA  FLAG
030A  8A                TXA
030B  48                PHA
030C  98                TYA
030D  48                PHA
030E  A9 00             LDA  #0       INIT FRAME ID
0310  85 32             STA  ID
0312  A5 30             LDA  ADR      SAVE BUF ADR
0314  85 37             STA  ABUF
0316  A5 31             LDA  ADR+1
0318  85 38             STA  ABUF+1

031A  A9 40       LD4   LDA  #$40     READ TIMING BIT
031C  85 34             STA  UNIT
031E  20 DB03           JSR  RBAR
0321  B0 F7             BCS  LD4
0323  A5 35             LDA  BAR          UNIT = 1/2 TIMING BIT
0325  4A                LSR  A
0326  85 34             STA  UNIT

0328  A0 00             LDY  #0       SEARCH FOR SYNC BYTE
032A  20 B303     LD6   JSR  RBIT
032D  B0 EB             BCS  LD4
032F  C0 16             CPY  #$16
0331  D0 F7             BNE  LD6

0333  20 A903     LD8   JSR  RBYT     READ CHECKSUM
0336  B0 E2             BCS  LD4
0338  84 39             STY  CKSM

033A  20 A903     LD10  JSR  RBYT     READ ID
033D  B0 DB             BCS  LD4
033F  C4 32             CPY  ID
0341  F0 0F             BEQ  LD12
0343  88                DEY
0344  C4 32             CPY  ID
0346  D0 D2             BNE  LD4
0348  E6 32             INC  ID           NEXT FRAME
034A  A5 30             LDA  ADR
034C  85 37             STA  ABUF
034E  A5 31             LDA  ADR+1
0350  85 38             STA  ABUF+1
0352  A5 37       LD12  LDA  ABUF         RESCAN
0354  85 30             STA  ADR
0356  A5 38             LDA  ABUF+1
0358  85 31             STA  ADR+1

035A  20 A903     LD14  JSR  RBYT     READ FRAME LENGTH
035D  B0 BB             BCS  LD4
035F  84 33             STY  LEN
0361  98                TYA
0362  F0 36             BEQ  LD24

0364  A5 3A       LD16  LDA  FLAG     SEE IF ABS OR REL
0366  F0 12             BEQ  LD18         IF REL
0368  20 A903           JSR  RBYT         ABS - READ LOAD ADDRESS
036B  B0 AD             BCS  LD4
036D  84 31             STY  ADR+1
036F  20 A903           JSR  RBYT
0372  B0 A5             BCS  LD4
0374  84 30             STY  ADR
0376  C6 33             DEC  LEN
0378  C6 33             DEC  LEN

037A  20 A903     LD18  JSR  RBYT     READ DATA
037D  B0 9B             BCS  LD4
037F  98                TYA
0380  A0 00             LDY  #0
0382  91 30             STA  (ADR),Y
0384  E6 30             INC  ADR
0386  D0 02             BNE  *+2
0388  E6 31             INC  ADR+1
038A  C6 33             DEC  LEN
038C  D0 EC             BNE  LD18

038E  A5 39       LD20  LDA  CKSM     CHECK CHECKSUM
0390  D0 88             BNE  LD4          IF ERROR
0392  A9 07             LDA  #07          OUTPUT CORRECT SIGNAL
0394  20 D902           JSR  TYPE
0397  38                SEC
0398  B0 80             BCS  LD4

039A  A5 39       LD24  LDA  CKSM     EOF READ
039C  D0 FA             BNE  LD4          IF CHECKSUM ERROR
039E  A9 07             LDA  #07          OUTPUT 'CORRECT' SIGNAL
03A0  20 D902           JSR  TYPE
```

```
03A3  68               PLA               RESTORE REGS
03A4  A8               TAY
03A5  68               PLA
03A6  AA               TAX
03A7  68               PLA
03A8  60               RTS          RETURN
                 *   ----
                 *   RBYT
                 *   ----
                 *
                 *
                 *    READ ONE BYTE FROM SCANNER
                 *    ADD BYTE TO CHECKSUM
                 *
                 *    EXIT: C(Y) = BYTE
                 *          CARRY = CLR IF BYTE READ
                 *                = SET IF END-OF-FRAME TIMEOUT
03A9  A2 08      RBYT  LDX  #8      SET BIT COUNT

03AB  20 BB03    BYT2  JSR  RBIT    READ BYTE
03AE  B0 DA            BCS  BYT9
03B0  CA               DEX
03B1  D0 F8            BNE  BYT2

03B3  98               TYA          ADD BYTE TO CHECKSUM
03B4  18               CLC
03B5  65 39            ADC  CKSM
03B7  85 39            STA  CKSM

03B9  18               CLC          RETURN
03BA  60         BYT9  RTS
                 *   ----
                 *   RBIT
                 *   ----
                 *
                 *    READ ONE BIT FROM SCANNER
                 *
                 *    EXIT: C(Y) = BYTE WITH BIT SHIFTED IN
                 *          CARRY = CLR IF BIT READ
                 *                = SET IF END-OF-FRAME TIMEOUT
03BB  20 DB03    RBIT  JSR  RBAR    READ BAR
03BE  B0 1A            BCS  BIT9

03C0  A5 34      BIT2  LDA  UNIT    SEE IF BAR>1.5*UNIT  (A 1 BIT)
03C2  4A               LSR  A
03C3  18               CLC
03C4  65 34            ADC  UNIT
03C6  38               SEC
03C7  E5 35            SBC  BAR
03C9  10 02            BPL  BIT4
03CB  46 35            LSR  BAR     ONE BIT - DIV BAR LEN IN HALF

03CD  0A         BIT4  ASL  A       SHIFT BIT INTO BYTE
03CE  98               TYA
03CF  2A               ROL  A
03D0  A8               TAY

03D1  A5 34            LDA  UNIT    COMPUTE NEW UNIT
03D3  18               CLC
03D4  65 35            ADC  BAR
03D6  4A               LSR  A
03D7  85 34            STA  UNIT

03D9  18               CLC          RETURN
03DA  60         BIT9  RTS
```

```
                 *   ----
                 *   RBAR
                 *   ----
                 *
                 *    READ BAR LENGTH
                 *
                 *    EXIT: C(BAR) = BAR COUNT
                 *          CARRY = CLR IF BIT READ
                 *                = SET IF END-OF-FRAME TIMEOUT

03DB  A9 00      RBAR  LDA  #0      CLEAR BAR COUNT
03DD  85 35            STA  BAR

03DF  A9 00      BAR1  LDA  #0      CLEAR SPACE COUNT
03E1  85 36            STA  SPAC

03E3  E6 36      BAR2  INC  SPAC    WAIT FOR SCANNER SET
03E5  F0 38            BEQ  BAR9
03E7  2C 12FC          BIT  SCNR
03EA  2C 12FC          BIT  SCNR
03ED  2C 12FC          BIT  SCNR
03F0  2C 12FC          BIT  SCNR
03F3  10 EE            BPL  BAR2

03F5  E6 35      BAR4  INC  BAR     WAIT FOR SCANNER CLEAR
03F7  F0 26            BEQ  BAR9
03F9  2C 12FC          BIT  SCNR
03FC  2C 12FC          BIT  SCNR
03FF  2C 12FC          BIT  SCNR
0402  2C 12FC          BIT  SCNR
0405  30 EE            BMI  BAR4

0407  A5 34            LDA  UNIT    SEE IF BAR>UNIT/4  (VALID DATA)
0409  4A               LSR  A
040A  4A               LSR  A
040B  38               SEC
040C  E5 35            SBC  BAR
040E  10 CF            BPL  BAR1

0410  A9 FD            LDA  #$FD    CHECK FOR SPACE STILL PRESENT
0412  85 36            STA  SPAC
0414  2C 12FC    BAR6  BIT  SCNR
0417  30 DC            BMI  BAR4
0419  E6 36            INC  SPAC
041B  D0 F7            BNE  BAR6

041D  18               CLC          NORMAL RETURN
041E  60               RTS

041F  38         BAR9  SEC          END-OF-FRAME TIMEOUT RETURN
0420  60               RTS
```

Notes

The 8080 or Z-80 Bar Code Loader Program

The 8080 or Z-80 program is able to use the registers in the computer to hold most of the program variables. The B, C, D and E registers contain the decoded byte, the unit width, the checksum, and the frame length, respectively. The HL register pair holds the buffer address. The only values which must be stored in memory are ABUF (buffer address at the beginning of the frame), ID (frame ID), and FLAG (the absolute or text format flag). The only programming "trick" used was to have the RBAR subroutine return to the calling program by jumping to the return sequences in RBIT (BIT7 for a normal return, and BIT9 for an end-of-frame timeout return). This saves a few bytes of code since both routines have to do similar cleanup operations before actually returning. The 8080 or Z-80 program was developed using a TDL Z-80 processor board running at 2 MHz. This program probably will not operate properly on a slow 8080 system because the bar counts will get too small to allow for good accuracy. Because of the inherent limitations of an 8080 microprocessor, the timing loops are about as fast as possible (which is not all that fast). This problem can be compensated for by scanning at a slower rate than would be used for an equivalent Z-80, 6502 or 6800 system.

LISTING NO. 3

```
;    ---------
;    LDA , LDR
;    ---------
;
;
;      SUBROUTINES TO LOAD DATA FROM BAR CODE SCANNER
;      INTO MEMORY.
;
;      LDA - LOADS ABSOLUTE BINARY DATA INTO MEMORY.
;            MEMORY ADDRESS IS CONTAINED IN DATA FRAME.
;
;      LDR - LOADS RELOCATABLE (E.G. ASCII) DATA NOT
;            ASSOCIATED WITH A MEMORY ADDRESS.
;            ENTER WITH H,L REGISTERS CONTAINING
;            ADDRESS OF WHERE TO STORE DATA.
;
;      REGISTER USAGE:
;
;           B - DECODED BYTE
;           C - UNIT WIDTH
;           D - CHECKSUM
;           E - FRAME LENGTH
;           HL- STORAGE ADDRESS
;
;           ALL REGISTERS EXCEPT H,L ARE SAVED ON ENTRY
;           AND RESTORED ON EXIT.  H,L WILL CONTAIN
;           ADDRESS OF LOCATION AFTER LAST DATA BYTE
;           LOADED INTO MEMORY.

                           PABS
1000                       LOC 01000H

F009                      TYPE=0F009H ;ADR OF ROUTINE TO TYPE A CHAR
0002                      SCNR = 2    ;I/O PORT OF SCANNER


1000 F5          LDA:   PUSH PSW    ;ABSOLUTE LOADER ENTRY POINT
1001 3E01               MVI  A,1
1003 C3 100C            JMP  LD2

1006 F5          LDR:   PUSH PSW    ;RELOCATABLE LOADER ENTRY POINT
1007 22 1116            SHLD ABUF
100A 3E00               MVI  A,0

100C 32 1119     LD2:   STA  FLAG
100F C5                 PUSH B
1010 D5                 PUSH D
1011 3E00               MVI  A,0
1013 32 1118            STA  ID

1016 0E28        LD4:   MVI  C,40   ;READ TIMING BIT
1018 CD 10E1            CALL RBAR
101B DA 1016            JC   LD4
101E 1F                 RAR
101F 4F                 MOV  C,A

1020 0600               MVI  B,0    ;SEARCH FOR SYNC BYTE
1022 CD 10B4     LD6:   CALL RBIT
1025 DA 1016            JC   LD4
1028 78                 MOV  A,B
1029 FE16               CPI  22
102B C2 1022            JNZ  LD6

102E CD 10A2     LD8:   CALL RBYT   ;READ CHECKSUM
1031 DA 1016            JC   LD4
1034 50                 MOV  D,B

1035 CD 10A2     LD10: CALL RBYT   ;READ ID
1038 DA 1016           JC   LD4
103B 3A 1118           LDA  ID          ;NEW FRAME OR RESCAN?
103E B8                CMP  B
103F CA 104D           JZ   LD12
1042 3C                INR  A
1043 B8                CMP  B
1044 C2 1016           JNZ  LD4         ;IF ILLEGAL ID
1047 32 1118           STA  ID          ;NEXT FRAME
104A 22 1116           SHLD ABUF
104D 2A 1116     LD12: LHLD ABUF        ;RESCAN

1050 CD 10A2     LD14: CALL RBYT   ;READ FRAME LENGTH
1053 DA 1016           JC   LD4
1056 58                MOV  E,B
1057 78                MOV  A,B
1058 FE00              CPI  0
105A CA 1093           JZ   LD24        ;IF EOF

105D 3A 1119     LD16: LDA  FLAG   ;SEE IF ABS OR REL
1060 FE00              CPI  0
1062 CA 1077           JZ   LD18        ;IF REL
1063 CD 10A2           CALL RBYT        ;IF ABS - READ ADDRESS
1068 DA 1016           JC   LD4
106B 60                MOV  H,B
106C CD 10A2           CALL RBYT
106F DA 1016           JC   LD4
1072 68                MOV  L,B
1073 7B                MOV  A,E
1074 3D                DCR  A
1075 3D                DCR  A
1076 5F                MOV  E,A

1077 CD 10A2     LD18: CALL RBYT   ;READ DATA
107A DA 1016           JC   LD4
107D 70                MOV  M,B
107E 23                INX  H
107F 7B                MOV  A,E
1080 3D                DCR  A
1081 5F                MOV  E,A
1082 C2 1077           JNZ  LD18

1085 7A          LD20: MOV  A,D    ;CHECK CHECKSUM
1086 FE00              CPI  0
1088 C2 1016           JNZ  LD4         ;IF ERROR
108B 0E07              MVI  C,07        ;OUTPUT 'CORRECT' SIGNAL
108D CD F009           CALL TYPE
1090 C3 1016           JMP  LD4

1093 7A          LD24: MOV  A,D    ;EOF READ
1094 FE00              CPI  0
1096 C2 1016           JNZ  LD4         ;IF CHECKSUM ERROR
1099 0E07              MVI  C,07        ;OUTPUT 'CORRECT' SIGNAL
109B CD F009           CALL TYPE
109E D1                POP  D           ;RETURN
109F C1                POP  B
10A0 F1                POP  PSW
10A1 C9                RET
```

```
                    ;     ----
                    ;     RBYT
                    ;     ----
                    ;
                    ;
                    ;     READ ONE BYTE FROM SCANNER
                    ;     ADD BYTE TO CHECKSUM
                    ;
                    ;     EXIT: C(B) = BYTE
                    ;           C(D) = CHECKSUM
                    ;           CARRY = CLR IF BYTE READ
                    ;                 = SET IF END-OF-FRAME TIMEOUT


10A2  3E08          RBYT: MVI  A,8     ;C(A) = BIT COUNT

10A4  CD 10B4       BYT2: CALL RBIT    ;READ BYTE
10A7  DA 10B3             JC   BYT9
10AA  3D                  DCR  A
10AB  C2 10A4             JNZ  BYT2        ;LOOP TO READ NEXT BIT

10AE  7A                  MOV  A,D     ;ADD BYTE TO CHECKSUM
10AF  80                  ADD  B
10B0  57                  MOV  D,A
10B1  37                  STC
10B2  3F                  CMC
10B3  C9            BYT9: RET
                    ;     ----
                    ;     RBIT
                    ;     ----
                    ;
                    ;
                    ;     READ ONE BIT FROM SCANNER
                    ;
                    ;     EXIT: C(B) = BYTE WITH BIT SHIFTED IN
                    ;           CARRY = CLR IF BIT READ
                    ;                 = SET IF END-OF-FRAME TIMEOUT

10B4  D5            RBIT: PUSH D       ;SAVE D,E
10B5  F5                  PUSH PSW     ;SAVE A,FLAGS
10B6  CD 10E1             CALL RBAR    ;READ BAR
10B9  DA 10DD             JC   BIT8
10BC  5F            BIT2: MOV  E,A     ;SEE IF BAR > 1.5*UNIT
10BD  79                  MOV  A,C
10BE  1F                  RAR
10BF  E67F                ANI  127
10C1  81                  ADD  C
10C2  BB                  CMP  E
10C3  F5                  PUSH PSW         ;(SAVE CARRY)

10C4  78                  MOV  A,B     ;SHIFT BIT INTO BYTE
10C5  17                  RAL
10C6  47                  MOV  B,A
10C7  F1                  POP  PSW     ; COMPUTE NEW UNIT
10C8  7B                  MOV  A,E
10C9  D2 10CE             JNC  BIT4
10CC  3F                  CMC
10CD  1F                  RAR
10CE  1F            BIT4: RAR
10CF  E67F                ANI  127
10D1  5F                  MOV  E,A
10D2  79                  MOV  A,C
10D3  1F                  RAR
10D4  E67F                ANI  127
10D6  83                  ADD  E
10D7  4F                  MOV  C,A         ; C(C) = UNIT
```

```
10D8  F1            BIT6: POP  PSW     ;NORMAL RETURN
10D9  D1            BIT7: POP  D
10DA  37                  STC
10DB  3F                  CMC
10DC  C9                  RET

10DD  F1            BIT8: POP  PSW     ;END-OF-FRAME TIMEOUT RETURN
10DE  D1            BIT9: POP  D
10DF  37                  STC
10E0  C9                  RET


                    ;     ----
                    ;     RBAR
                    ;     ----
                    ;
                    ;
                    ;     READ BAR LENGTH
                    ;
                    ;     EXIT: C(A) = BAR COUNT
                    ;           CARRY = CLR IF BAR READ
                    ;                 = SET IF END-OF-FRAME TIMEOUT


10E1  D5            RBAR: PUSH D       ;SAVE D,E
10E2  1E00                MVI  E,0     ;CLEAR BAR COUNT

10E4  1600          BAR1: MVI  D,0     ;CLEAR SPACE COUNT

10E6  14            BAR2: INR  D       ;WAIT FOR SCANNER SET
10E7  CA 10DE             JZ   BIT9
10EA  DB02                IN   SCNR
10EC  FE00                CPI  0
10EE  F2 10E6             JP   BAR2

10F1  1C            BAR4: INR  E       ;WAIT FOR SCANNER CLR
10F2  CA 10DE             JZ   BIT9
10F5  DB02                IN   SCNR
10F7  FE00                CPI  0
10F9  FA 10F1             JM   BAR4
10FC  79                  MOV  A,C     ;SEE IF BAR > UNIT/4 (VALID DATA)
10FD  E6FC                ANI  252
10FF  1F                  RAR
1100  1F                  RAR
1101  BB                  CMP  E
1102  D2 10E4             JNC  BAR1

1105  1603                MVI  D,3     ;CHECK FOR SPACE STILL PRESENT
1107  DB02          BAR6: IN   SCNR
1109  FE00                CPI  0
110B  FA 10F1             JM   BAR4
110E  15                  DCR  D
110F  C2 1107             JNZ  BAR6
1112  7B                  MOV  A,E     ;NORMAL RETURN
1113  C3 10D9             JMP  BIT7


                      ;   ------------
                      ;   DATA STORAGE
                      ;   ------------

1116  0000          ABUF: .WORD 0      ;BUF ADDR AT BEGINNING OF FRAME
1118  00            ID:   .BYTE 0      ;FRAME ID
1119  00            FLAG: .BYTE 0      ;ABS/REL FLAG
```

Notes

Implementation and Checkout Procedure

1. Verify the hardware connections to the scanner. The "wand" unit and electronics employed must be level sensitive, translating reflectance of a white paper into a data value of 0 on its output line, translating reflectance of a black (fully inked) paper into a data value of 1 on its output line. (Some commercial point of sale scanners produce edge timing information in the form of pulses which occur when light changes to dark and vice versa. These scanners are unusable with the programs given here.) The output line of the scanner electronics should be connected to the high order bit of the 8 bit input port used by the programs of listings 1 to 3.

2. Using the manual methods (ie: keyboard and monitor program, toggle switches, etc.) of your system, enter one of the programs from listing 1 to listing 3. Modify the program's hardware dependent address constants to suit your system's hardware constraints. If you use a processor other than a 6800, 6502, 8080 or Z-80, then use the flowcharts of figure 4 and examples of listings 1 to 3 to create a new loader program for your processor.

3. Verify the operation of the loader program by using one pass of the data contained in figure 2b and comparing the results to the data listed in the figure. For those who use listings 1 to 3 for the program, most problems will probably be found in the area of making the hardware dependent address changes. More general debugging may be needed if a new program is coded for a different processor. Use the Text Entry Procedure (see separate box) for this checkout operation.

4. With the loader's operation verified, save it on your system's mass storage device; make sure the cassette or floppy disk copy is verified against the memory image of the program, and make redundant copies if you require that degree of safety.

Using The Bar Code Loader Algorithm

Text Entry Procedure

This procedure is used whenever reading bar code texts which have been encoded using the "text" format of figure 3a. In this format, the bar code copy is used to define an address independent block of data which can be placed in an arbitrary buffer in memory. Typical types of data involved are character source texts of applications programs, character data files in general and relocatable object code files which will be processed further by appropriate linking loaders, etc.

1. Make sure that your bar code loader program has been correctly loaded into a scratch area of memory, and that the hardware is all set up. Set up of the hardware includes initialization of the scanner input port if this is required, as in the case of those who use PIA (Motorola 6820) input ports.

2. Set the initial value of the pointer ABUF. For the 6800 program of listing 1, this is accomplished by loading the index (X) register prior to entry. In the 6502 program of listing 2, this is accomplished by initializing the variable ADR which is at location hexadecimal 30 in memory in listing 2. For the 8080 or Z-80 program of listing 3, this is accomplished by initializing the H and L register pair with the starting memory pointer. ABUF should be set so that during the course of the loading operation it will not conflict with the memory location of the loader program itself, or for that matter, any other program which you want to preserve.

3. Physically prepare for the first scan by laying the bar codes on a flat surface, obtaining a ruler or straight edge which is longer than the longest frame of bars by several inches, and positioning yourself comfortably.

4. Start the bar code loader program by calling the LDR entry point from your monitor.

5. For each frame of the bar code text being read, position the ruler so that the wand will scan with its aperture centered directly over the bars. Use guide marks (built in or added by yourself) on the wand head to set the ruler position. Then, with a steady hand, move the wand down the line of bars starting from about one half to three fourths of an inch before the beginning of the frame, and continuing at a steady rate until the end of the frame has been scanned. If the frame was successfully read, the terminal device of your system will sound the "bell" code (a bell on Teletypes, or tone of some form on CRT terminals). When you have received a correct read acknowledgement go on to the next frame of the text.
   If no acknowledgement is heard, there was a timeout or checksum error and the frame was incorrectly read. Repeat the same frame, after checking the ruler position, your scanning technique, etc. This feedback interactively teaches you how to correctly position the ruler and wand; from our own experience, once the technique is practiced a bit, nearly every frame will be correctly positioned and read.

6. When the last frame has been read with a zero length and zero checksum, end of file is determined and the program loader will return to the calling point. If no end of file frame is found in the bars, return can also be effected by restarting the system in your usual manner.

7. This has read the data into memory starting at the initial value of ABUF. What is done with the bar code originated data depends on the documentation accompanying the program or other text which you have just read.

A General Bar Code Loader Algorithm

Absolute Entry Procedure

This procedure is used whenever reading bar code texts which have been encoded using the simple "absolute" loader format of figure 3b. In this format, the bar code data of each frame begins with a two byte destination address for the data, high order byte first. This form is generally used with absolute object code of simple programs which are compiled for fixed addresses in memory. Such programs are generally ready to run upon completion of the loading process.

1. Make sure that your bar code loader program has been correctly loaded into a scratch area of memory, and that the hardware is all set up. Hardware set up should include initialization of the scanner input port if necessary. Using the documentation of the program being input, verify that the absolute addresses encoded in the bar code file are consistent with available memory areas in your system.

2. Physically prepare for the first scan by laying the bar codes on a flat surface, obtaining a ruler or straight edge which is longer than the longest frame of bars by several inches, and positioning yourself comfortably.

3. Start the bar code loader program by calling the LDA entry point from your monitor.

4. For each frame of the bar code text being read, position the ruler so that the wand will scan with its aperture centered directly over the bars. Use guide marks (built in or added by yourself) on the wand head to set the ruler position. Then, with a steady hand, move the wand down the line of bars starting from about one half to three fourths of an inch before the beginning of the frame, and continuing at a steady rate until the end of the frame has been scanned. If the frame was successfully read, the terminal device of your system will sound the "bell" code (a bell on Teletypes, or tone of some form on CRT terminals). When you have received a correct read acknowledgement go on to the next frame of the text.

   If no acknowledgement is heard, there was a timeout or checksum error and the frame was incorrectly read. Repeat the same frame, after checking the ruler position, your scanning technique, etc. This feedback interactively teaches you how to correctly position the ruler and wand; from our own experience, once the technique is practiced a bit, nearly every frame will be correctly positioned and read.

5. When the last frame has been read with a zero length and zero checksum, end of file is determined and the program loader will return to the calling point. If no end of file frame is found in the bars, return can also be effected by restarting the system in your usual manner.

6. This has loaded data in regions of your system's memory which are encoded within the bar code text. Proceed to use the data as specified in the documentation accompanying the bar codes; for example, if the data is a program loaded in absolute form, call or jump to the appropriate entry point address.

A Note About Bar Codes . . .

Our intent in making Paperbytes^{TM} software available in bar code form is to provide a method of conveying machine readable information from documentation to the memories and mass storage of a user's system on a one time basis. We suggest that the user of software obtained in this manner should locally record the data on the mass storage devices of his system after the data has been scanned from the printed page. The Paperbytes^{TM} bar code representations provide a standardized means of obtaining the data, but they cannot be compared to the convenience of local mass storage devices such as floppy disks, digital cassettes or audio cassettes. Thus if repeated use of the software obtained from bar code is anticipated, we recommend that the user make a copy on some form of magnetic medium.

Bar codes are the newest form of machine readable data representation. They are used in all Paperbyte TM software products in BYTE magazine articles and self contained book publications and combine efficiency of space, low cost, and ease of data entry with the need for mass produced machine readable representations of software. Bar codes were originally used for product identification in inventory control and supermarket checkout applications. Today, because of their direct binary representation of data, they are an ideal computer compatible communications medium. In the application of bar codes to software distribution (such as Paperbyte TM books and articles), the use of a simple but reliable optical scanning wand and an appropriate program provides a convenient means for the user to acquire software.

PAPERBYTE ™ – An Exciting New Way To Distribute Software

One of the most common problems for users and suppliers of personal computer software is the need for product distribution in a form which is helpful to the user, low in cost, tolerant of errors in production use, and free of the need for expensive highly specialized peripherals. One solution, conceived in detail by Walter Banks of the Computer Communications Network Group at the University of Waterloo, Ontario, Canada, is the use of bar code patterns prepared on a computer controlled phototypesetter. A bar code is a linear array of printed bars of varying width which encodes digital data as alternating patterns of black ink and white paper. By using a ruler as a guide, an inexpensive hand held "wand" scanning unit converts the bar patterns into a time varying logic level signal. This time varying binary value can then be interpreted by a program which understands the format of the bars.

The purpose of this pamphlet is to present the decoding algorithm which was designed by Ken Budnick of Micro-Scan Associates at the request of BYTE Publications Inc. The text of this pamphlet was written by Ken, and contains the general algorithm description in flow chart form plus detailed assemblies of program code for 6800, 6502 and 8080 processors. Individuals with computers based on these processors can use the software directly. Individuals with other processors can use the provided functional specifications and detail examples to create equivalent programs.

0-07-008856-x